Task-based fast multipole method for clusters of multicore processors

Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Martin Khannouz, Luka Stanisic

Task-based fast multipole method for clusters of multicore processors

Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Martin Khannouz, Luka Stanisic

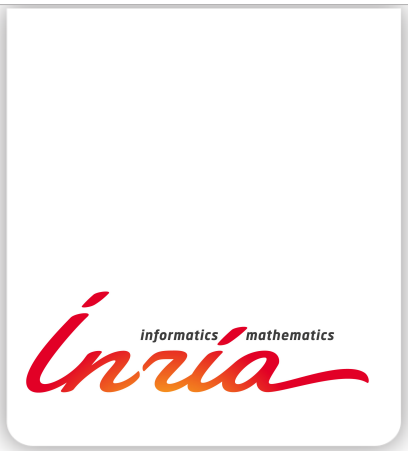
RESEARCH REPORT

N° 8970

October 2016

Project-Teams HiePACS and STORM

ISSN 0249-6399 ISRN INRIA/RR--8970--FR+ENG



Task-based fast multipole method for clusters of multicore processors

Emmanuel Agullo*, Bérenger Bramas†, Olivier Coulaud*,
Martin Khannouz*, Luka Stanisic*

Project-Teams HiePACS and STORM

Research Report n° 8970 — version 2 — initial version October 2016 —
revised version March 2017 — 21 pages

Abstract: Most high-performance, scientific libraries have adopted hybrid parallelization schemes - such as the popular MPI+OpenMP hybridization - to benefit from the capacities of modern distributed-memory machines. While these approaches have shown to achieve high performance, they require a lot of effort to design and maintain sophisticated synchronization/communication strategies. On the other hand, task-based programming paradigms aim at delegating this burden to a runtime system for maximizing productivity. In this article, we assess the potential of task-based fast multipole methods (FMM) on clusters of multicore processors. We propose both a hybrid MPI+task FMM parallelization and a pure task-based parallelization where the MPI communications are implicitly handled by the runtime system. The latter approach yields a very compact code following a sequential task-based programming model. We show that task-based approaches can compete with a hybrid MPI+OpenMP highly optimized code and that furthermore the compact task-based scheme fully matches the performance of the sophisticated, hybrid MPI+task version, ensuring performance while maximizing productivity. We illustrate our discussion with the ScalFMM FMM library and the StarPU runtime system.

Key-words: high performance computing (HPC), fast multipole method, FMM, cluster, multicore processor, runtime system, hybrid parallelization, task-based programming, MPI, OpenMP

* Inria, France, Email: surname.name@inria.fr
† Max Planck Computing and Data Facility (RZG), Garching, Germany, Email: Berenger.Bramas@mpcdf.mpg.de

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Méthode des multipôles rapide à base de tâches pour des clusters de processeurs multicoeurs

Résumé : La plupart des bibliothèques scientifiques très performantes ont adopté des parallélisations hybrides - comme l'approche MPI+OpenMP - pour profiter des capacités des machines modernes à mémoire distribuée. Ces approches permettent d'obtenir de très hautes performances, mais elles nécessitent beaucoup d'efforts pour concevoir et pour maintenir des stratégies de synchronisation/communication sophistiquées. D'un autre côté, les paradigmes de programmation à base de tâches visent à déléguer ce fardeau à un moteur d'exécution pour maximiser la productivité. Dans cet article, nous évaluons le potentiel de la méthode des multipôles rapide (FMM) à base de tâches sur les clusters de processeurs multicœurs. Nous proposons deux types de parallélisation, une première approche hybride (MPI+Tâche) à base de tâches et d'appels à MPI pour gérer explicitement les communications et la deuxième uniquement à base de tâches où les communications MPI sont implicitement postées par le moteur d'exécution. Cette dernière approche conduit à un code très compact qui suit le modèle de programmation séquentiel à base de tâches. Nous montrons que cette approche rivalise avec le code hybride MPI+OpenMP fortement optimisé et qu'en outre le code compact atteint les performances de la version hybride MPI+Tâche, assurant une très haute performance tout en maximisant la productivité. Nous illustrons notre propos avec la bibliothèque FMM ScalFMM et le moteur d'exécution StarPU.

Mots-clés : Calcul haute performance, méthode multipôles rapide, FMM, cluster, architecture multicœur, moteur d'exécution, parallélisation hybride, programmation à base de tâches, MPI, OpenMP

Contents

List of Algorithms

1 Introduction

The hardware complexity of computers used in scientific computing has continuously increased since the first parallel machines. In particular, distributed-memory machines now almost systematically rely on multicore processors. As a consequence, most parallel, scientific libraries have been redesigned to ensure high-performance on such architectures. The most common approach consists in developing two-level hybrid parallelization schemes that cope with the hardware hierarchy. The Message Passing Interface (MPI) paradigm is most often employed to allow for inter-node parallelism while a lower footprint threading system such as OpenMP is in charge of the parallelization within a node. Although this hybrid MPI+OpenMP approach has proven to be very efficient for a wide range of scientific applications, it tends to lack versatility and is hard to maintain. For these reasons, the high-performance computing (HPC) community has been investigating a lot whether higher-level programming paradigms could achieve a competitive performance.

The fast multipole method (FMM) [1] has been proposed by Leslie Greengard and Vladimir Rokhlin in 1987 to reduce the complexity of N-body simulations. While the accurate calculation of the motions of N particles interacting via gravitational or electrostatic forces was considered to require $\theta(N^2)$ operations (one for each pair of particles), the FMM does so in $\theta(N)$ operations by using multipole expansions (mass or net charge, dipole moment, quadrupole, and so forth) to approximate the effects of a distant group of particles on a local group. Considered among the top ten algorithms of the twentieth century [2], the FMM has been used a lot in the twenty-first century by the HPC community, winning ACM Gordon Bell Prize awards for applications in turbulence [3] and blood flow [4] simulations, reaching beyond petascale for the latter. These astonishing breakthroughs have been achieved with hybrid parallelization schemes relying on MPI, OpenMP and possibly CUDA.

On the other hand, a number of studies have been conducted to assess whether the FMM could be designed using alternative programming paradigms. [5] and [6] for instance assessed FMM implemented on top of PGAS and Charm++, respectively. Among alternative approaches, the design of task-based FMM on top of runtime systems has received a special attention, yet at a more modest scale, on either a (single) multicore [7, 8, 9] or a heterogeneous node [10]. While achieving equally good or higher performance than lower-level approaches, the main benefit of these task-based schemes is that they can be written as extremely compact codes following a sequential task-based programming model, namely the sequential task flow (STF) paradigm.

In this article, we assess the potential of task-based FMM on distributed memory machines, more specifically on clusters of multicore processors. We first describe a hybrid MPI+Task FMM parallelization. In this scheme, MPI communications are submitted as tasks so that they can be finely interleaved with computational tasks by the runtime system. However, communications must still be explicitly written by the programmer, leading to a relatively complex and error-prone code. Therefore, we propose a second algorithm where the MPI communications are implicitly handled by the runtime system. This latter approach yields a compact code, following a sequential task-based paradigm.

Besides the design and implementation of two new task-based codes (sections 3.1 and 3.2) for exploiting distributed-memory machines (which are the first task-based FMM codes for distributed-memory machines to the best of our knowledge), the main contribution of this study is to show how a very competitive performance can be achieved with a compact sequential task-based algorithm on distributed memory machines for one of the most important algorithms in scientific computing. We stress that the focus of this paper is not how to design a scalable runtime system (we rely on [11] for that) but how modern task-based runtime systems can be used to design fully-featured, compact and scalable numerical codes. Although demonstrated at a rel-

atively modest scale (960 cores), especially with respect to the aforementioned previous studies, we think that these results are important enough to draw more attention from the parallel and distributed processing community on the design of fully-featured, scalable numerical libraries on top of task-based runtime systems.

The rest of the paper is organized as follows. We present the FMM algorithm in Section 2 together with an MPI+OpenMP parallelization scheme (Section 2.2) and a task-based scheme for single node multicore architectures from [9] (Section 2.3). We then describe how we extended this task-based solution to distributed-memory parallelization using explicit or implicit communication approaches in sections 3.1 and 3.2, respectively. We finally assess the efficiency of these algorithms in Section 4 before concluding (Section 5).

2 Background

The FMM is a widely used algorithm to reduce the quadratic complexity of dual-based interactions introduced in [1]. It has been the object of many studies and we refer the reader to [12] for a detailed overview. In the following, we only present the key ingredients of the method together with a high-level description of the algorithm so that one can understand the principles of this algorithm and the main data-structures it relies on.

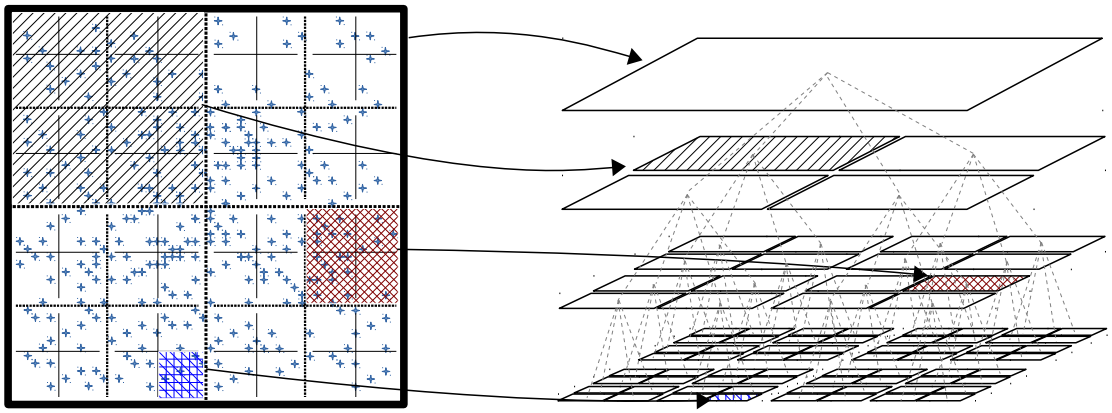


Figure 1: 2D hierarchical decomposition of the plane (left) and corresponding quadtree (right).

Initially introduced in the context of N particles interacting via gravitational forces, the FMM has since been used in a wide range of applications such as astrophysical simulations, molecular dynamics, the boundary element method, radiosity in computer-graphics and fluid dynamics. The crucial part of the FMM for reducing the complexity from $\theta(N^2)$ operations (one computation for each pair of particles) to $\theta(N)$ consists in approximating interactions between distant particles in a hierarchical manner. The key-point of the FMM algorithm is to approximate the far-field - the interactions between far-apart particles - while maintaining a desired accuracy. The interactions between close particles still remain computed with a direct Particle to Particle (P2P) method, while the far-field is processed using a tree-based algorithm instead. A tree data structure is built from a recursive subdivision of the space performed during an initialization step. We note h the height of the tree, which is also the number of recursions plus one. In the current study we use the term *octree* in a generic manner to refer to the FMM tree even if the type of the tree (quadtree, octree, ...) is related to the dimension of the problem. Figure 1 is an example of an octree and shows the relationship between the spatial decomposition and the data

structure. It also illustrates how each cell represents its descendants composed of its children and sub-children.

2.1 Sequential FMM

Algorithm 1: Sequential FMM

```
1  Function FMM(tree, kernel)
2      // Near-field
3      P2P(tree, kernel);
4      // Far-field
5      P2M(tree, kernel);
6      for l = tree.height-2 → 2 do
7          M2M(tree, kernel, l);
8      for l = 2 → tree.height-2 do
9          M2L(tree, kernel, l);
10         L2L(tree, kernel, l);
11     M2L(tree, kernel, tree.height-1);
12     L2P(tree, kernel);
13 Function M2M(tree, kernel, level)
14     foreach cell cl in tree.cells[level] do
15         kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
```

To compute the interactions between all particles inside the simulation box, the FMM algorithm proceeds in four steps, namely the direct pass, the upward pass, the transfer pass and the downward pass. The former step handles spatially close particles (or *near-field*) while the other three steps perform the computations between distant particles (or *far-field*). Algorithm 1 presents an overview of the sequential pseudo-code and we now detail the role of each operator. In the direct pass/*near-field*, the FMM computes the interactions between spatially close particles with a classical/direct particle-to-particle operator (P2P, line 3 in Algorithm 1). This operation is applied at the bottom of the tree between each leaf and its corresponding neighbors. The key of the FMM algorithm for achieving a lower complexity than a classical direct computation is that these P2P operations, which consist of the exact same operations as in a classical direct computation, are restricted to clusters of spatially close particles. In the FMM, the interactions between distant particles are then indeed computed hierarchically in the *far-field*. The general principle consists of using cells to transfer the information across the tree, having each cell composed of a multipole and a local parts. In the electrostatic context, the multipole (M) of a given cell represents an approximation of the charge of its descendants. On the other hand, the local part (L) of a cell c represents the potential field due to all cells well-separated of cell c that will be applied to the descendants of c.

The far-field starts with the upward pass that aggregates the physical values of the particles from bottom to top. First, this is done at the leaf level using the Particle to Multipole ($P2M$, line 5), and then at upper levels by the Multipole to Multipole ($M2M$, lines 6-7) operator. After this operation, each cell hosts the contributions of its descendants in its multipole part. More precisely, this M2M operator corresponds to a computation of a multipole expansion at a level l based on the multipole expansions of the child cells at level $l+1$ as shown at lines 13-15 (for a matter of conciseness, we do not detail the other operators). In the transfer pass, the Multipole to Local ($M2L$, lines 8-9 and 11) operator is applied between each cell and its corresponding interaction list at all levels. The interaction list for a given cell c at level l is composed by the children of the neighbors of c's parent that are not direct neighbors/adjacent to c. After

the transfer pass, the local part of all the cells are filled with contributions. The downward pass aims to apply these contributions to the particles. In this pass, the local contributions are propagated from top to bottom with the Local to Local ($L2L$, lines 8-10) operator, and applied to the particles with the Local to Particle ($L2P$, line 12) operator. After these far-field operations, the particles have received their respective far contributions. Figure 2 illustrates how the contributions from different points of the grid are applied to a given leaf.

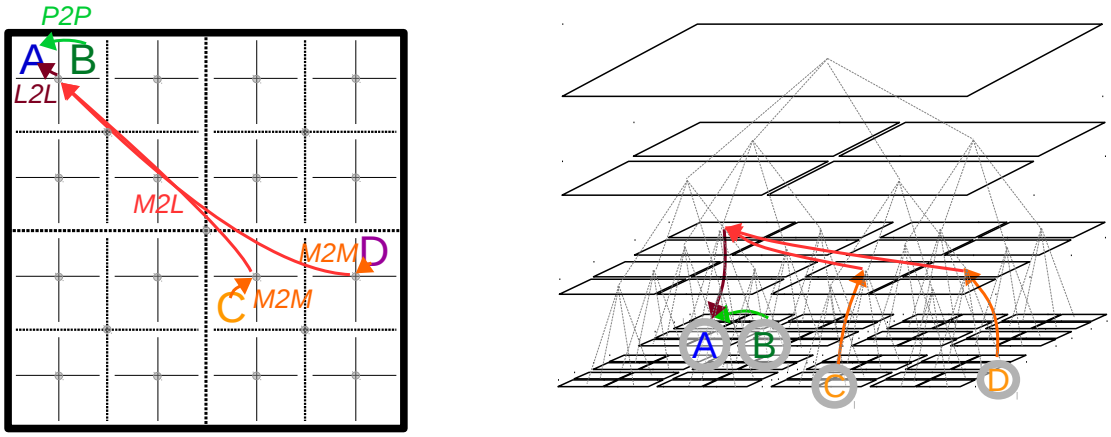


Figure 2: The contributions from leaves B, C and D are applied to A using the different operators depending on their respective distance to A.

2.2 MPI+OpenMP3 FMM (hierarchical clusters)

The first parallel implementations of the FMM for distributed memory machines were designed on top of MPI [13, 14, 15, 16]. The advent of multicore processors and GPUs led the community to combine MPI for inter-node parallelism with either Posix or OpenMP threads within the node [17, 18, 19], possibly enhanced with accelerator directives [18, 20, 21]. We now present the hybrid MPI+OpenMP FMM algorithm implemented in ScalFMM. Although each FMM library has its own specificities and optimizations, this algorithm is representative of advanced FMM hybrid parallelization schemes and we refer to [22] for a more comprehensive overview. The entire FMM is divided into sub-trees mapped onto the different processes. In the particular case of ScalFMM (and thus in particular in the experiments conducted in this paper), the partitioning is done using the Morton space filling curve [23] such that each process owns a Morton index interval. We currently rely on a balancing heuristic ensuring that each process gets approximately attributed either the same number of particles or the same number of leaves.

Shaded lines in Algorithm 2 show how the sequential M2M operator FMM (lines 13-15 in Algorithm 1) is turned into a hybrid MPI+OpenMP algorithm. Note that we do not report the main FMM function because it remains unchanged, i.e. exactly the same as lines 1-12 in Algorithm 1. Once the threads are created (line 2 in Algorithm 2), one thread is dedicated to the communication stage (lines 4-9) while the other threads compute the local M2M (lines 11-13). Working at level l, the thread in charge of communications sends the children of the first cell if needed (lines 5-6) and receives the children of the last cell if needed (line 7-8). If all the communications have been completed while there is still some remaining computation to be performed, this thread may then join other threads to perform actual computation. Indeed, the

Algorithm 2: Hybrid MPI+OpenMP3 M2M (hierarchical clusters). The main FMM function remains as lines 1-12 in Algorithm 1.

```
1  Function M2M(tree, kernel, level)
2      #pragma omp parallel
3          // One thread dedicated to communication
4          #pragma omp single nowait
5              if must_send_first_children(tree, level) then
6                  MPI_ISend( tree.cells[level].first..children.multipole );
7              if must_recv_children_last_parent(tree, level) then
8                  MPI_IRecv(&recv_children);
9              MPI_Waitall();
10         // All threads compute local cells
11         #pragma omp for schedule(dynamic, 10) nowait
12         foreach cell cl in tree.local_cells[level] do
13             kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
14         // Wait completion of both communication and computation
15         #pragma omp barrier
16         // Compute received data if needed
17         if must_recv_children_last_parent(tree, level) then
18             kernel.M2M(tree.cells[level].last, recv_children);
```

barrier is only posted at line 15, ensuring that the communication and computation are completed when the threads potentially apply the contributions received from their children (lines 17-18).

We refer to this algorithm as hybrid MPI+OpenMP3 in the sequel as it relies on OpenMP3 features, yet not employing dependent tasks from the revision 4 of the standard.

2.3 Task-based FMM (single node)

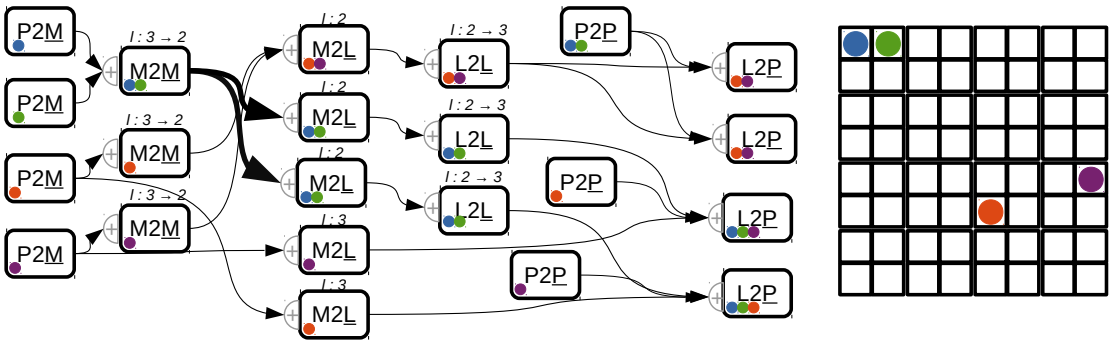


Figure 3: FMM DAG associated with the configuration depicted in Figure 2.

Task-based programming paradigms consist in designing an algorithm as a set of tasks and dependences between them. A directed acyclic graph (DAG) can be conveniently used to represent such an algorithm, tasks being associated with vertices and dependences with edges. DAGs have long been used in computer science to represent parallel programs. They are for instance commonly used at the instruction level in compilation [24] or, on the other side of the spectrum,

at the job level in grid computing where they are commonly referred to as workflows [25]. With the advent of multicore processors and accelerators as main processing units, the increase of hardware complexity led the parallel and distributed communities to also investigate its usage at an intermediate level (tasks of approximately one millisecond) for programming supercomputers. Figure 3 shows for instance the FMM DAG associated with the particle distribution depicted in Figure 2.

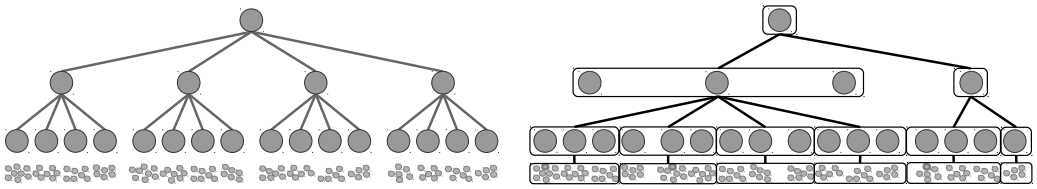


Figure 4: From a quadtree (left) to a group tree (right), $n_g = 3$.

One of the challenges of task-based programming is to ensure an appropriate granularity of the tasks. They must be fine enough so that many tasks can get executed concurrently but coarse enough so that the dependences between tasks can be handled at runtime without a significant overhead. In the particular case of the FMM, [8] showed that designing tasks as baseline FMM operators led to a significant overhead of the runtime system management. They introduced a new data structure, the group tree, to reduce this overhead. The subsequent algorithm operates on groups of n_g (the *group size*) consecutive leaves or cells (following Morton indexing) as illustrated in Figure 4.

Another challenge consists in (correctly) encoding the DAG. [26] proposed a task-based model, namely the *Parameterized Task Graph* (PTG), where both tasks and dependences (list of predecessors and successors) between them consist of parametrized expressions. This model was initially thought to describe regular algorithms (dense linear algebra factorizations) and was later extended to support irregular algorithms [27, 28]. Let us once again consider the configuration and associated DAG represented in figures 2 and 3, respectively. Because they have no predecessors, P2M and P2P tasks can be immediately processed. When, for instance, the P2M task associated with the orange particle is completed, its M2M and M2L successors can be triggered since they have no other predecessor. In this model, the local expansion of the DAG traversal can thus be performed with the unique knowledge of the related P2M successors and M2M predecessors, without requiring any global view of the DAG. This property is exploited by the PaRSEC [28] runtime system to perform decentralized DAG unrolling, ensuring an excellent scalability to the numerical task-based libraries that have been developed on top of it [29]. However, the expression of the dependences is very challenging.

On the contrary, the *Sequential Task Flow* (STF) model allows for the automatic computation of dependences based on sequential consistency. Tasks are inserted in a sequential (valid) order as one would write a sequential algorithm. Shaded lines in Algorithm 3 show how the sequential FMM (Algorithm 1) is turned into an STF algorithm. Instead of executing a kernel (such as the M2M kernel at line 15 in Algorithm 1), the execution of the kernel is submitted to the runtime system. The exact code for performing this submission depends on the runtime system that is employed but the principle is common to all runtime systems supporting the STF model such as StarPU [30], OmpSs [31], Quark [32] and many others. Without loss of generality, we consider the particular syntax of StarPU as it will be employed to illustrate the discussion in our experiments (Section 4). The submission is performed with the *starpu_insert_task*() instruction (line 18 in Algorithm 3). Based on the data access modes (READ, WRITE or

Algorithm 3: STF FMM (single node)

```
1  Function FMM(tree, kernel)
2      // Near-field
3      P2P_insert_tasks(tree, kernel);
4      // Far-field
5      P2M_insert_tasks(tree, kernel);
6      for l = tree.height-2 → 2 do
7          M2M_insert_tasks(tree, kernel, l);
8      for l = 2 → tree.height-2 do
9          M2L_insert_tasks(tree, kernel, l);
10         L2L_insert_tasks(tree, kernel, l);
11     M2L_insert_tasks(tree, kernel, tree.height-1);
12     L2P_insert_tasks(tree, kernel);
13     // Wait for completion
14     starpu_wait_all_tasks();
15 Function M2M_insert_tasks(tree, kernel, level)
16     foreach cell_group cl in tree.cell_groups[level] do
17         foreach cell_group cl_child in cl.sub_groups do
18             starpu_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                 cl_child.multipole);
```

READ_WRITE), the runtime system automatically computes the dependences with previously submitted tasks. Unlike compiler theory where the dependence analysis is done statically and on scalars [24], the dependences are evaluated at runtime and at the granularity of tasks operating on groups of cells (the tree now being a group tree). The submission of tasks being asynchronous, we have to eventually wait for their completion (line 14).

3 Task-based FMM for hierarchical clusters

We now propose the design of two new task-based FMM algorithms for distributed memory machines. The first algorithm (Section 3.1) aims at exploiting the hierarchy of the cluster by combining explicit MPI communications for inter-node management and a task-based scheme within the node. While this scheme is designed to deliver very high performance by combining the benefits of explicit MPI communications together with the DAG parallelism inherent to task-based schemes, it leads to another hybrid scheme, whose synchronizations are as complex to handle as for the MPI+OpenMP3 scheme discussed above. This is why we then investigate whether a competitive performance can be delivered with a compact STF algorithm (Section 3.2). The key idea of this second approach is to automatically infer MPI communications from the DAG [33, 34, 11].

3.1 Hybrid MPI+Task FMM for hierarchical clusters

The first task-based FMM algorithm for distributed memory machines we propose aims at exploiting the FMM DAG information within a node. The tree is built in parallel with the same procedure and mapping as for the above hybrid MPI+OpenMP3 algorithm, except that it operates on a group data-structure. In the MPI+OpenMP3 FMM code, one thread is explicitly dedicated to the communications (lines 4-9 in Algorithm 2 for the M2M operator). In the MPI+Task algorithm, the communications are submitted to the runtime system with the

Algorithm 4: Hybrid MPI+Task M2M (hierarchical clusters). The main FMM function remains as lines 1-16 in Algorithm 3.

```
1 Function M2M_insert_tasks_and_comms(tree, kernel, level)
2     foreach cell_group cl in tree.cell_groups[level] do
3         foreach cell_group cl_child in cl.sub_groups do
4             starpu_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                cl_child.multipole);
5         foreach cell_group cl_child in cl.remote.sub_groups do
6             starpu_mpi_irecv_detached(cl_child.multipole);
7             starpu_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                cl_child.multipole);
8         if cl has remote parent then
9             starpu_mpi_isend_detached(cl.multipole)
```

starpu_mpi_irecv/isend_detached instructions (lines 6/9 in Algorithm 4). The runtime system handles the actual communication requests to the MPI layer when the data becomes available, transparently ensuring an advanced interleaving of the communications with computational tasks. For instance, StarPU-MPI [35] dedicates one thread to ensure the progress of communications while other threads perform computational tasks, submitted at lines 4 and 7 in Algorithm 4 in the case of the M2M operator.

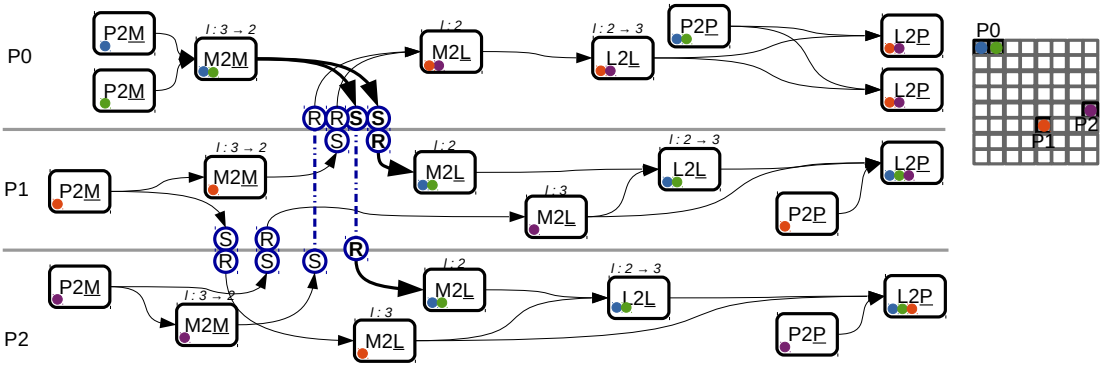


Figure 5: FMM DAG associated with the configuration depicted in Figure 2 scattered onto three processes. Communication tasks are represented with R (reception) and S (send) symbols.

We illustrate the behavior of our algorithm considering once again the configuration depicted in Figure 2. We assume that leaves A (blue) and B (green) are mapped on process P0, whereas leaves C (orange) and D (purple) are mapped on processes P1 and P2, respectively, as shown in Figure 5 (right). Once P0 has processed the M2M task related to the blue and green cells, it sends (rightmost bold S tasks in Figure 5 as well as line 9 in Algorithm 4) the resulting multipole values to P1 and P2. Conversely, P1 and P2 post the matching receives (rightmost bold R tasks in Figure 5) within the hybrid MPI+Task M2L algorithm (not reported here). This approach requires the programmer to maintain symbolic information such that each node is able to post the communication. For instance, the processes do not have to maintain a complete view of all the data hosted by the other processes but at least the ones needed to perform their own receives and sends.

3.2 STF FMM for hierarchical clusters

3.2.1 STF (without data extraction)

The hybrid MPI+Task algorithm presented above strongly reduces the burden of pipelining the computational tasks with the communications compared to the hybrid MPI+OpenMP3 scheme, which had to rely on an advanced usage of OpenMP3 features to achieve a similar behavior. However, it still requires to explicitly post the communications, leading once again to a hybrid MPI+X code whose complexity may be error-prone. We now consider a second algorithm which fully delegates the communication management to the runtime system so that they become transparent at the level of the numerical algorithm. The runtime can indeed infer the communications from the data dependences. For instance, assuming the distribution and mapping shown in Figure 5, the communication tasks related to the blue and green cells occurring on P0 (send) and P1 or P2 (receive), respectively, can be readily inferred from the dependences between related M2M and M2L tasks in the global DAG (bold edges in Figure 3). As a consequence, one may write a compact STF code while achieving high performance on distributed memory machines. Successive studies [33, 34, 11] assessed (and refined) this protocol showing its potential on relatively regular numerical algorithms. However, none of them assessed it on an irregular, numerical algorithm such as we do in the present study with the FMM.

Algorithm 5: STF FMM (hierarchical clusters)

```
1  Function FMM(tree, kernel)
2      // Register and map the multipole/local/particles groups
3      map_data(tree);
4      // Near-field
5      P2P_mpi_insert_tasks(tree, kernel);
6      // Far-field
7      P2M_mpi_insert_tasks(tree, kernel);
8      for l = tree.height-2 → 2 do
9          M2M_mpi_insert_tasks(tree, kernel, l);
10     for l = 2 → tree.height-2 do
11         M2L_mpi_insert_tasks(tree, kernel, l);
12         L2L_mpi_insert_tasks(tree, kernel, l);
13     M2L_mpi_insert_tasks(tree, kernel, tree.height-1);
14     L2P_mpi_insert_tasks(tree, kernel);
15     // Wait for completion
16     starpu_wait_all_tasks();
17 Function M2M_mpi_insert_tasks(tree, kernel, level)
18     // Same tasks inserted on all procs
19     foreach cell_group cl in tree.cell_groups[level] do
20         foreach cell_group cl_child in cl.sub_groups do
21             starpu_mpi_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                  cl_child.multipole);
```

We report the entire FMM algorithm following this STF model in Algorithm 5. Shaded lines show how the baseline STF FMM operating on a single node (Algorithm 3) is turned into an STF algorithm for a distributed memory machine, emphasizing the compactness of the resulting code. First, the data must be registered to the runtime system and mapped on the different processes (line 3). During this operation, the different runtime system instances save the mapping for all the data such that they know where each piece of data is located initially. After this registration and mapping step, the algorithm is (almost) exactly the same as the STF code operating on a

single node (Algorithm 3). The only exception is that the *starpu_mpi_insert_task*() primitive is used instead of *starpu_insert_task*(). Consistently with [11] (on which we rely), each process fully unrolls the whole DAG. The *starpu_mpi_insert_task*() primitive then uses the mapping information (previously registered at line 3) to automatically decide whether the task must actually be processed (in the case the process is the owner), whether a receive must be posted (the process is the owner of the task but not of its predecessors), and whether a send must be posted (the process is the owner of the task but not of a successor). Using this simple protocol yields the exact same DAG as the one in Figure 5 but communication tasks (S and R) are now implicitly inserted by the runtime system.

In the hybrid MPI+Task algorithm, each MPI process was only aware of the tasks it is in charge of. On the contrary, in this STF strategy, each process traverses the whole DAG. Of course, only tasks actually mapped on a given process are executed by this process. Furthermore, the decision for discarding a task that is not mapped on the considered MPI process is almost immediate. However, the sum of such decisions on a given process may prevent the DAG to be unrolled sufficiently fast to deliver enough concurrency to all available CPU cores. On a modern supercomputer equipped with multicore processors, one process per node is indeed typically instantiated. Within this process, one thread is in charge of unrolling the DAG, one thread handles the communication, and so-called worker threads, or *workers* for short, perform the actual tasks. The optimum number of workers may vary depending on the algorithm and on the target platform. For instance, it may be worth dedicating one core for unrolling the DAG and handling communications by setting a number of workers lower than the number of available CPU cores. We will study the impact on performance of such a set up for the proposed STF FMM algorithm in the experimental section.

3.2.2 STF with data extraction

Algorithm 6: STF M2M with extract (hierarchical clusters). The main FMM function remains as lines 1-16 of the STF without extract (Algorithm 5).

```
1  Function M2M_mpi_insert_tasks_extract(tree, kernel, level)
2      // Same tasks inserted on all procs
3      foreach cell_group cl in tree.cell_groups[level] do
4          foreach cell_group cl_child in cl.sub_groups do
5              proc_cl ← proc_owner(cl.multipole) ;
6              proc_child ← proc_owner(cl_child.multipole) ;
7              if proc_cl == proc_child then
8                  cl_child_multipole ← cl_child.multipole;
9              else
10                 cl_child_extracted ← init_data_on_node(proc_child);
11                 starpu_mpi_insert_task(extract, WRITE, cl_child_extracted, READ,
                     cl_child.multipole);
12                 cl_child_multipole ← init_data_on_node(proc_cl);
13                 starpu_mpi_insert_task(restore, WRITE, cl_child_multipole, READ,
                     cl_child_extracted);
14             starpu_mpi_insert_task(kernel.M2M_group, READ_WRITE, cl.multipole, READ,
                 cl_child_multipole);
```

The MPI+OpenMP3 implementation from Section 2.1 explicitly builds the MPI messages by packing the cells/leaves. On the contrary, the task-based algorithms we propose rely on groups

to increase the granularity; as a result, they communicate groups of n_g cells/particle-leaves. Therefore, complete groups of n_g cells are sent even if a remote task needs only few elements inside a group. Therefore, this scheme may lead to communicate (much) more data than needed. Reducing the granularity can limit the amount of data exchanged in the messages. However, it would also amplify the overhead of the runtime system as the number of tasks to take care of increases, which may lessen the overall performance. We will assess this trade-off in Section 4.2.1.

Alternatively, we can extract the useful part of the data to be communicated. This mechanism is equivalent to classical packing schemes in MPI. In a task-based paradigm, it can be designed as in Algorithm 6 (illustrated for the M2M operator only for a matter of conciseness). Before inserting a computational task, we test if its dependences are mapped to the same process (line 7). In this specific example, we check the owners of the parent (*cl.multipole*) and child groups (*cl_ child.multipole*). If they are not mapped to the same process, *cl_ child.multipole* will have to be communicated. We create (line 10) a temporary data (*cl_ child_ extracted*) on node *proc_ child* (which accesses the data in READ mode) and insert a new task in charge of performing the extraction (line 11) of the useful part of the data. This new task does not need any communication since *cl_ child.multipole* and *cl_ child_ extracted* have been mapped to the same process. On the other hand, we allocate (line 12) a data (*cl_ child_ multipole*) on node *proc_ cl* (which accesses the data in WRITE mode) and we insert a task to reshape the group in a valid form (line 13). This is the task that will trigger the communication, as *cl_ child_ multipole* and *cl_ child_ extracted* are not mapped to the same process. The benefit over the previous scheme is that the data exchanged (*cl_ child_ extracted* instead of *cl_ child.multipole*) is smaller. Our protocol is valid thanks to the assumption that the runtime system computes a task where the data in write mode is mapped (which is the case in our set up). We refer to this variant as *STF with extract*. Its advantage over STF without extract is that the amount of communication is potentially strongly decreased. On the other hand, the extraction has itself a cost (symbolic computation on indexes must be performed). We will assess this overall trade-off in Section 4.2.2.

4 Experiments

4.1 Experimental Setup

4.1.1 Hardware and software

The experiments have been performed on the *Draco* computer from the MPCDF. It is composed of Intel Xeon E5-2698 processors with 32 cores (2.3 GHz each). Each node has 128 GB of main memory and the nodes are inter-connected with InfiniBand FDR14. We use between 1 and 30 nodes (960 cores). We use gcc 5.4, Intel MKL 11.3, Intel MPI 5.1.3 and StarPU 1.2. We exploit the commutativity of FMM operations (see [9] for details). For the purpose of the present study, we have implemented all three proposed task-based FMM algorithms (namely, the hybrid MPI+Task, STF without extract and STF with extract schemes, corresponding to algorithms 4, 5, 6 respectively) on top of StarPU-MPI [35]. Both STF algorithms furthermore rely on a submission window [11] to limit the pressure on the MPI layer. All these features we rely on are embedded in StarPU 1.2. We assess both configurations with 31 and 32 workers per node (see discussion in Section 3.2.1). As discussed in Section 2.2, we use the hybrid MPI+OpenMP3 from [22] as reference. This code can be viewed as an optimized, state-of-the-art MPI+OpenMP3 FMM and it is based on the same data-structures and numerical kernels (described below) as the task-based schemes.

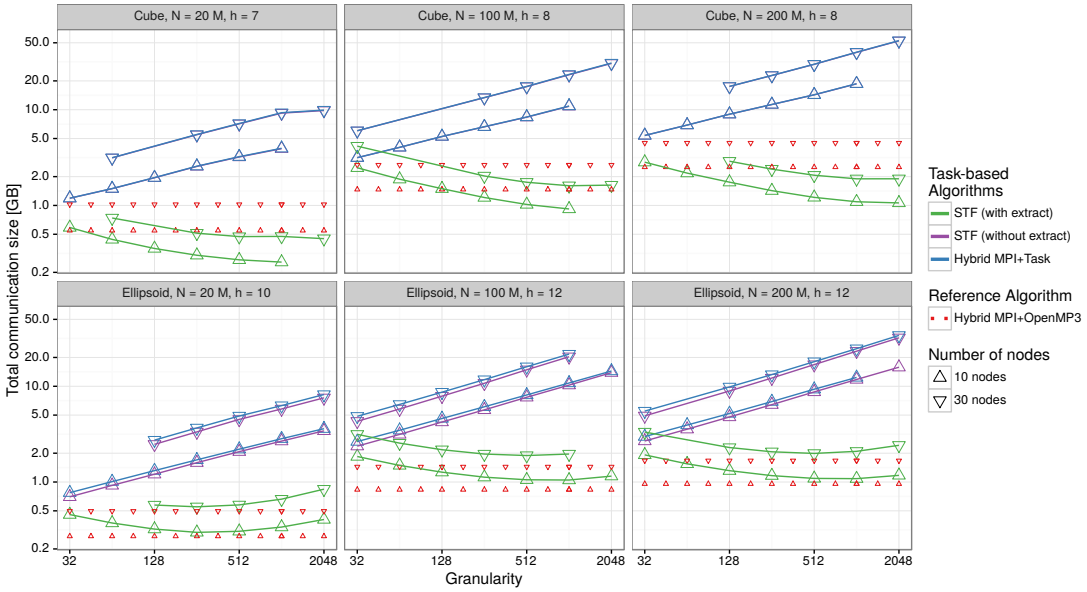


Figure 6: Impact of the granularity (log2 scale) on the total volume of communication (GB, log10 scale). Note that the STF (without extract) and the hybrid MPI+Task perfectly overlap with each other in the cube case (and almost overlap in the ellipsoid case).

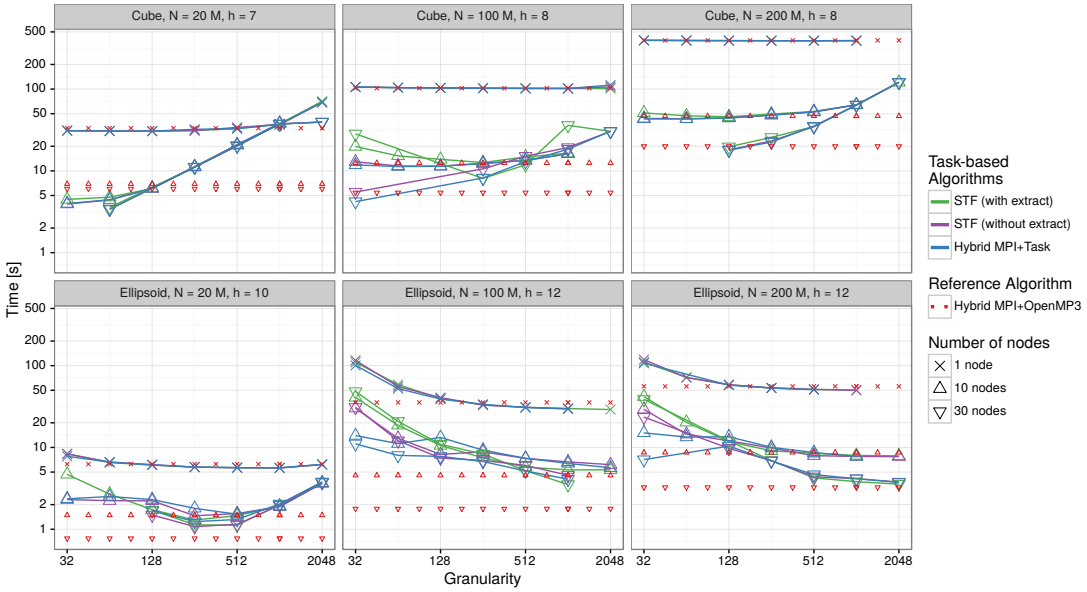


Figure 7: Impact of the granularity (log2 scale) on the execution time (seconds, log10 scale). All 32 workers per node are turned on for task-based algorithms.

4.1.2 Numerical test cases

We use the Chebyshev FMM derived from [36] with an accuracy of 10^{-6}. We consider two types of particle distributions. *Cube (volume)* distributions are composed of particles uniformly distributed in a unit cube, leading to a regular octree and a high and well balanced amount of work for each cell/leave. On the other hand, *ellipsoid (surface)* distributions are composed of particles distributed on the surface of an ellipsoid with a high density at the poles, leading to an irregular octree and a highly variable amount of work associated with the nodes of the octree and an overall low arithmetic intensity. As a consequence, those ellipsoid test cases are much more challenging to parallelize. Each type of particle distribution is assessed with a number N of particles equal to 20 (very hard to efficiently parallelize on 30 nodes), 100 (intermediate), or 200 (reasonable to parallelize on 30 nodes, at least for a cube distribution) millions. The height h of the tree is chosen so that it minimizes the single-node execution (assessed with the MPI+OpenMP3 code). The particle-to-process mappings of task-based schemes have been adapted to follow the one of the MPI+OpenMP3 reference.

4.2 Results

4.2.1 Effect of the granularity

Figure 6 shows the impact of the granularity n_g on the total communication volume. The hybrid MPI+Task (blue) and STF without extract (purple) schemes induce a communication volume significantly higher than the hybrid MPI+OpenMP3 reference algorithm (red), especially when a coarse granularity is used. On the contrary, as discussed in Section 3.2.2, the STF with extract scheme (green) remarkably reduces this volume. Because the description of groups relies on different data structures than non-grouped schemes, the symbolic information that is exchanged differs between the STF (with extract) and hybrid MPI+OpenMP3 schemes, and one can observe this (slight) impact on the communication volume. Figure 7 shows that this decrease of the volume of communication does not however improve the overall execution timings. Indeed, StarPU dedicates one thread for ensuring the progress of communications so that they can be efficiently interleaved with computation. Regarding the discussion in Section 3.2.2, we conclude that, on this particular platform, for all test cases we have assessed, the (yet significant) decrease of communication does not pay off the cost of extraction.

4.2.2 Efficiency

Figure 8 shows the parallel efficiency of the different strategies normalized with the fastest execution on one node. The results where the number of workers corresponds to the number of cores (circle points •) correspond to the above results with variable granularity, and we report the performance obtained with optimum granularity. As motivated in Section 3.2.1, we also assess the performance of our task-based algorithms when turning off one worker per node (diamond points ⌖). Overall, we observe that all three task-based schemes (green, purple, blue plots) are very competitive with the reference MPI+OpenMP3 scheme (red dotted line). Furthermore, turning-off one worker (diamond points ⌖) for dedicating one core to the progress (of communications and task insertions) ensures a much more robust performance. The most significant benefits arise for the smallest (20 M particles – left) or irregular (ellipsoid distributions – bottom) test cases, which are extremely challenging to efficiently parallelize. Interestingly, in this setup, both the STF without extract (purple) and hybrid MPI+Task (blue) proposed algorithms consistently outperform the reference hybrid MPI+OpenMP3 (dotted red), yet highly optimized, code.

These results show that FMM can achieve very high performance with a task-based design

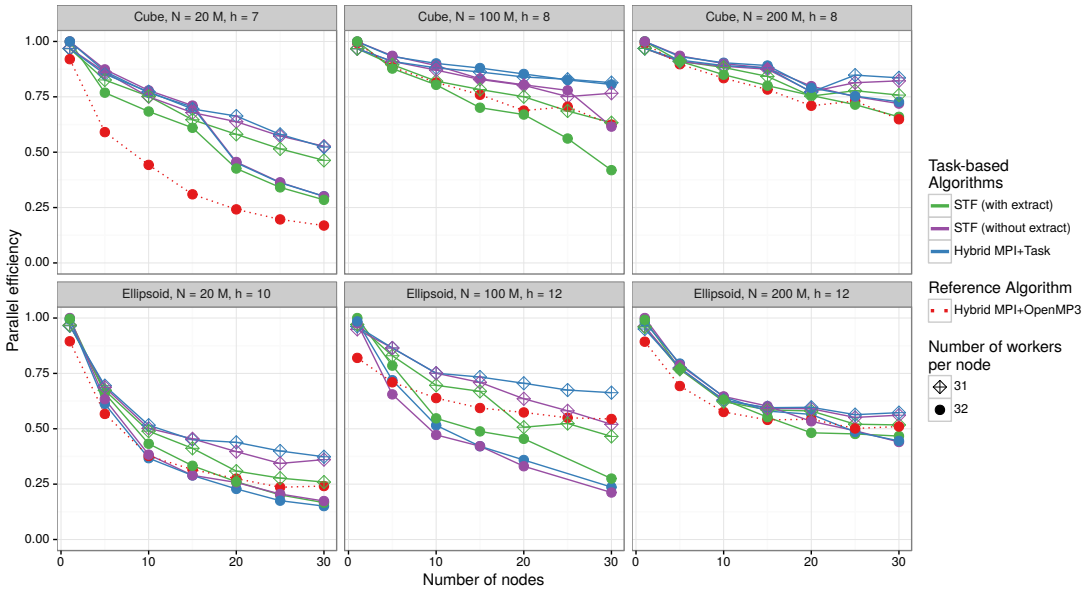


Figure 8: Parallel efficiency (normalized with the fastest algorithm on a single node).

on a cluster of multicore processors. We can furthermore observe that the cost for automatically inferring the communications is relatively low. Indeed, the hybrid MPI+Task (blue diamonds) and STF without extract (purple diamonds) algorithms follow the same communication pattern, except that, in the STF case, the runtime system automatically infers them. This leads to a much more compact code (as shown in Section 3.2.1), while ensuring an excellent efficiency, only marginally lower than with the explicit communication scheme, as shown in Figure 8.

5 Conclusion

We have proposed two task-based FMM algorithms for clusters of multicore nodes. Similarly to most high-performance codes targeting such architectures, the first algorithm (MPI+Task) is a hybrid approach that aims at combining the benefits of explicit MPI communications together with the DAG parallelism inherent to task-based schemes. The second algorithm (STF) is a very compact code that delegates the burden of synchronizations and communications to the runtime system (StarPU in the present study). Both approaches have been implemented within the ScalFMM library to have a fair comparison between the proposed task-based codes and the optimized, original MPI+OpenMP3 code. All implementations indeed rely on the same computational kernels and data distribution schemes and differences only come from the parallelization and code expression. We showed that a very competitive performance can be achieved with a compact sequential task-based algorithm (STF) on a cluster of multicore processors. Although demonstrated at a relatively modest scale (960 cores), we hope that these results will encourage the parallel processing community to consider the design of fully-featured, scalable numerical libraries on top of task-based runtime systems.

We considered a variant of the second algorithm that extracts the exact data to be communicated in order to reduce the volume of communication (STF with extract). On very irregular particle distribution that yields lower arithmetic intensity, such as the ellipsoid test case, this

extraction mechanism needs to operate at a relatively coarse granularity, which ensures efficient usage of the kernels when operating on irregular data, while maintaining a reasonable volume of communications. Still, when turning off one worker per node to dedicate one core for the progress (of communications and of task insertions), we showed that communications get fully overlapped and that it was not worth performing the extractions in those configurations. We plan to study the opportunity to delegate this extraction scheme to the runtime system and devise a dynamic strategy for automatically triggering (or not) the extraction. Finally, in this paper we wanted to study the ability of task-based schemes to compete with hybrid MPI+OpenMP3 approaches for exploiting hierarchical clusters. For this reason, we ensured that the data mapping of all our task-based schemes match the one of the hybrid MPI+OpenMP3 code. However, task-based algorithms (and in particular the STF model) provide the opportunity to perform any data and task mapping, as the mapping is orthogonal to the algorithm design in this model. We plan to exploit this property to explore new mapping strategies ensuring load-balancing in both near- and far-fields, which could significantly benefit to irregular particular distributions.

Acknowledgment

Computations were performed on the HPC system "Draco" of the Max Planck Computing and Data Facility (RZG). We thank the StarPU development team for their support as well as Grégoire Pichon for proofreading an early draft of this paper.

References

[1] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of computational physics*, vol. 73, no. 2, pp. 325–348, 1987.

[2] F. Sullivan and J. Dongarra, "Guest editors' introduction: The top 10 algorithms," *Computing in Science & Engineering*, vol. 2, no. 1, pp. 22–23, 2000.

[3] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji, "42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence," in *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. ACM, 2009.

[4] A. Rahimian, I. Lashuk, S. K. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, and A. Shringarpure, "Petascale direct numerical simulation of blood flow on 200K cores and heterogeneous architectures," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, November 2010.

[5] J. Milthorpe, A. P. Rendell, and T. Huber, "PGAS-FMM: Implementing a distributed fast multipole method using the X10 programming language," *Concurrency and Computation: Practice and Experience*, 2013.

[6] M. AbdulJabbar, R. Yokota, and D. Keyes, "Asynchronous Execution of the Fast Multipole Method Using Charm++," 2014.

[7] H. Ltaief and R. Yokota, "Data-driven execution of fast multipole methods," *Concurrency and Computation: Practice and Experience*, no. September, pp. 1935–1946, 2013.

[8] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for multicore architectures," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. C66–C93, 2014.

[9] E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset, "Bridging the gap between OpenMP 4.0 and native runtime systems for the fast multipole method," Inria, Research Report RR-8953, Mar. 2016. [Online]. Available: https://hal.inria.fr/hal-01372022

[10] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for heterogeneous architectures," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 9, pp. 2608–2629, 2016.

[11] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," Research Report RR-8927, Jun. 2016. [Online]. Available: https://hal.inria.fr/hal-01332774

[12] L. Greengard and V. Rokhlin, "A new version of the Fast Multipole Method for the Laplace equation in three dimensions," *Acta Numerica*, vol. 6, pp. 229–269, 1997.

[13] M. S. Warren and J. K. Salmon, "Astrophysical N-body simulations using hierarchical tree data structures," in *Proc. of the 1992 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '92. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 570–576.

[14] S. Ogata, T. J. Campbell, R. K. Kalia, A. Nakano, P. Vashishta, and S. Vemparala, "Scalable and portable implementation of the fast multipole method on parallel computers," *Computer Physics Communications*, vol. 153, no. 3, pp. 445–461, 2003.

[15] J. Kurzak and B. M. Pettitt, "Massively parallel implementation of a fast multipole method for distributed memory machines," *Journal of Parallel and Distributed Computing*, vol. 65, no. 7, pp. 870–881, 2005.

[16] F. A. Cruz, M. G. Knepley, and L. A. Barba, "PetFMM-a dynamically load-balancing parallel fast multipole library," *International Journal for Numerical Methods in Engineering*, vol. 85, no. 4, pp. 403–428, 2011.

[17] O. Coulaud, P. Fortin, and J. Roman, "Hybrid MPI-Thread Parallelization of the Fast Multipole Method," in *Parallel and Distributed Computing, 2007 (ISPDC)*, 2007, p. 52.

[18] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc, "Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on.* IEEE, 2010, pp. 1–12.

[19] D. Malhotra, A. Gholami, and G. Biros, "A volume integral equation stokes solver for problems with variable coefficients," in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE Press, 2014, pp. 92–102.

[20] T. Ishiyama, K. Nitadori, and J. Makino, "4.45 pflops astrophysical n-body simulation on k computer: The gravitational trillion-body problem," in *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 5:1–5:10.

[21] J. Choi, A. Chandramowlishwaran, K. Madduri, and R. Vuduc, "A CPU: GPU Hybrid Implementation and Model-Driven Scheduling of the Fast Multipole Method," in *GPGPU-7, Workshop on General Purpose Processing Using GPUs*, 2014, pp. 64–71.

[22] B. Bramas, "Optimization and parallelization of the boundary element method for the wave equation in time domain," PhD Thesis, Université de Bordeaux, Feb. 2016. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01306571

[23] M. S. Warren and J. K. Salmon, "A hashed oct-tree n-body algorithm," in *Proc. of the 1993 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 12–21.

[24] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[25] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," *ACM Sigmod Record*, vol. 34, no. 3, pp. 44–49, 2005.

[26] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *System Sciences, 1995. Proc. of the Twenty-Eighth Hawaii International Conference on*, vol. 2, Jan 1995, pp. 113–122 vol.2.

[27] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar, "Concurrent collections," *Sci. Program.*, vol. 18, no. 3-4, pp. 203–217, Aug. 2010.

[28] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, 2013. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2013.98

[29] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Hérault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. Yarkhan, and J. J. Dongarra, "Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA," in *Proc. of the 25th IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11), PDSEC 2011*, Anchorage, United States, May 2011, pp. 1432–1441. [Online]. Available: https://hal.inria.fr/hal-00809680

[30] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 2011.

[31] A. Duran, J. M. Perez, R. M. Ayguadé, E. amd Badia, and J. Labarta, "Extending the OpenMP tasking model to allow dependent tasks," in *OpenMP in a New Era of Parallelism, 4th International Workshop, IWOMP 2008*. West Lafayette, IN: Lecture Notes in Computer Science 5004:111-122, May 12-14 2008.

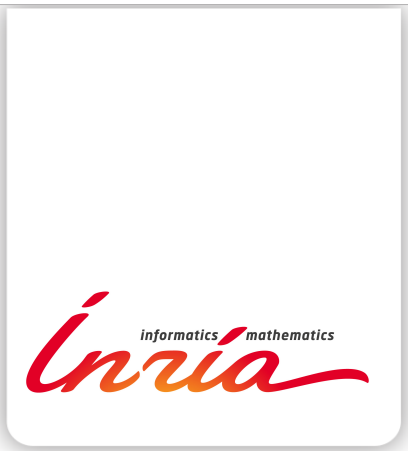
[32] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK users' guide: QUeueing And Runtime for Kernels," Innovative Computing Laboratory, University of Tennessee, Tech. Rep. ICL-UT-11-02, April 2011, http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf.

[33] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, "A high-productivity task-based programming model for clusters," *Concurrency and Computation: Practice and Experience*, 2012.

[34] A. YarKhan, "Dynamic task execution on shared and distributed memory architectures," Ph.D. dissertation, University of Tennessee, 2012.

[35] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms," in *ICPADS*, Shanghai, China, Dec. 2010.

[36] W. Fong and E. Darve, "The black-box fast multipole method," *Journal of Computational Physics*, vol. 228, no. 23, pp. 8712 – 8725, 2009.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399